Algotithm Complexity

by Xingyu Yang

Before we start to learn about DSAs (algorithm and Datastructures), we need to first understand the concept of algorithm, data structure, and their complexity, which is the main focus of this section.

Contents

§1. Defining Algorithm and Data Structure

It is always important to understand the basic concepts before we start to learn about the more advanced topics. In this part, we will first introduce the concept of algorithm and data structure, and then we will discuss the complexity of algorithms and data structures.

§1.1. Algorithm

What is algorithm? The answer may varie even among the computer scientists, since definition can be quite board as long as they fit into most part of the big picture. However, we can define algorithm as a set of instructions that are used to solve a problem or perform a task. In other words, an algorithm is a step-by-step procedure that is used to solve a problem or perform a task. Formally, in rigorous mathematical language, we have the following definition.

Definition 1.1.1 (Algorithm).
Algorithm is a **finite** sequence of **well-defined**, **computer-implementable** instructions, typically to solve a class of problems or to perform a computation.

Formally, an algorithm is a function $A : \Sigma^* \to \Sigma^*$ that is computable by a Turing machine M, where Σ is a finite alphabet, and Σ^* represents the set of all finite strings over Σ. The function A must satisfy the following properties:

1. **Finiteness**: M must halt for all valid inputs within a finite number of steps.
2. **Well-defined**: For every input $x \in \Sigma^*$, the output $A(x)$ is uniquely determined.
3. **Effectiveness**: The function A can be implemented using a finite set of instructions that can be executed mechanically (e.g., by a Turing machine or an equivalent model of computation).

To elaborate, an algorithm must has a stop condition, which means it must halt for all valid inputs within a finite number of steps, otherwise, it will be an infinite loop. Moreover, the output of the algorithm must be uniquely determined for every input, which means the algorithm must be deterministic. Finally, the algorithm must be implementable using a finite set of instructions that can be executed mechanically, such as a Turing machine or an equivalent model of computation (lambda calculus or recursive function).

Example (Macthing String with Turing Machine).
Consider an example of a trivial algorithm that determine whether a given string is consist of only character '0'.

This algorithm is runed by a simple Turing machine that recognizes the language

$$L = \{0^n \mid n \geq 0\}.$$

This means the machine accepts any string containing only the character `0` (and no any other symbols).

To make such a machine, we simply need to set up the following rules. First, we determine the **possible states** of the machine:
1. q_0: the initial state
2. q_{accept}: the accepting state
3. q_{reject}: the rejecting state

Then, we define the **transition rules** for the machine:
1. If we are in state q_0 and we see a `0`, we stay in state q_0 and move right.
2. If we are in state q_0 and we see a blank symbol, this means that the end of string has been reached, and we move to state q_{accept} and move right, return accept.
3. If we are in state q_0 and we see any other symbol, we move to state q_{reject} and move right, return reject.

Throughout the excecution, we keep track of the current state, the current symbol, and the position of the head on the tape. At the beginning, the tape is filled with blanks, and the head is positioned at the beginning of the input string. The machine will keep running until it reaches the accepting or rejecting state. On top of that, the way of transition is **deterministic** by the rules we defined.

The following Python code shows the implementation of the Turing machine that recognizes the language $L = \{0^n \mid n \geq 0\}$. This is optional for those who want to take a more challenging approach to understand the concept of algorithm and Turing machine. While for most, you may only need to understand how a turing machine embodies the characteristics of an algorithm without knowing pricise implementation.

```python
class TuringMachine:
    def __init__(self, transitions, start_state, accept_state, reject_state):
        """
        transitions: dict with keys  -> (state, symbol)
                     and values -> (new_state, new_symbol, direction)
        """
        self.transitions = transitions
        self.state = start_state
        self.accept_state = accept_state
        self.reject_state = reject_state
        self.tape = ["_"] * 100  # Blank tape (bigger just for safety)
        self.head = 50           # Start in the middle
    def run(self, input_string):
        """Loads the input and simulates the Turing Machine."""
        # 1. Load the input onto the tape
        for i, symbol in enumerate(input_string):
            self.tape[self.head + i] = symbol

        # 2. Keep running until we accept or reject
        while self.state not in (self.accept_state, self.reject_state):
            current_symbol = self.tape[self.head]

            if (self.state, current_symbol) in self.transitions:
                new_state, new_symbol, direction = self.transitions[(self.state,
current_symbol)]

                # Write the new symbol
                self.tape[self.head] = new_symbol
                # Move to the new state
                self.state = new_state
                # Move head left or right
                if direction == "R":
                    self.head += 1
                else:  # "L"
                    self.head -= 1
            else:
                # No valid transition => reject
                self.state = self.reject_state

        # 3. Check whether we've accepted or rejected
        return "Accepted" if self.state == self.accept_state else "Rejected"

# --------------------------------------------------------
# Transitions for L = { all strings of 0's }
# --------------------------------------------------------
# Explanation:
# (q0, '0') -> (q0, '0', R): Keep scanning right if we see a '0'
# (q0, '_') -> (q_accept, '_', R): If we see blank, we accept
#             (means only zeros so far, done reading)
# Anything else => reject (by default: no transition => reject)

transitions = {
    ("q0", "0"): ("q0", "0", "R"),     # see a zero, stay in q0, move right
    ("q0", "_"): ("q_accept", "_", "R") # see a blank, accept
}
def create_tm() -> TuringMachine:
    return TuringMachine(
        transitions=transitions,
        start_state="q0",
        accept_state="q_accept",
        reject_state="q_reject"
    )

def reset() -> None:
    # using global variable only for convenience here, not a good practice
    global tm
    tm = create_tm()
```

```
# Test Cases
tm = create_tm()
print("Input ''        ->", tm.run(""))        # "" (empty string) => Accepted
reset()
print("Input '0'       ->", tm.run("0"))       # "0" => Accepted
reset()
print("Input '000'     ->", tm.run("000"))     # "000" => Accepted
reset()
print("Input '001'     ->", tm.run("001"))     # Contains a '1' => Rejected
reset()
print("Input '111'     ->", tm.run("111"))     # All ones => Rejected
```

This interesting example has brought us to the intersection of computability theory and algorithm theory. The concept of an algorithm is not just about the code we write but also about the theoretical models of computation, such as the Turing machine, which forms the foundation of modern computer science. Moreover, we can see that the computational model intertwines with the concept of an algorithm and its three defining characteristics. An intriguing fact is that whether a problem is computable remains an open question in computer science—there exists no algorithm that can determine whether the execution of a given algorithm (modeled as a Turing machine) will halt. This is known as the Halting Problem, one of the most famous undecidable problems in computer science.

§1.2. Data Structure

§2. Naive Notion of Complexity

§2.1. Time Complexity

§2.2. Space Complexity